Consuming an Smart Contract from a simple WebApp

1st Daniel Santiago Muñoz
Dept of ELectric and Industrial einginiering
UNAL
Bogotá, Colombia
danmunozbe@unal.edu.co

***Abstract*—This paper presents the implementation of a decentralized application using smart contracts deployed on the Ethereum blockchain. The development process includes contract coding in Solidity, local testing with Ganache, and frontend interaction via JavaScript and HTML with MetaMask as wallet.**

***Index Terms*—Smart Contract, Ganache, Solidity, Ethereum, MetaMask**

I. Introduction

As discussed in the cybersecurity class, we were introduced to the concept and implementation of smart contracts. By using Ether as a medium of control, these contracts enable a decentralized system of transactions, making interference more difficult. This paper presents the process of implementing such technologies using Remix to host the smart contract, Ganache to simulate a local blockchain with test accounts, and key JavaScript/HTML components for front-end interaction with MetaMask for signatures.

II. Creating the Smart Contract

The smart contract was developed using Remix IDE, an online development environment specifically designed for writing, compiling, and deploying Solidity code on the Ethereum platform.

In this contract, we define two private variables: `name` and `age`. The goal is to allow reading and writing of these values through publicly accessible functions. The contract serves as a simple example of persistent storage and controlled access within a blockchain environment.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract PersonStorage {
    string private name;
    uint256 private age;

    function setName(string memory _name)
        public {
        name = _name;
    }

    function setAge(uint256 _age) public {
        age = _age;
    }

    function getName() public view returns
        (string memory) {
        return name;
    }

    function getAge() public view returns
        (uint256) {
        return age;
    }
}
```

Listing 1. Smart contract: PersonStorage

III. Setting Up Ganache

To interact with the smart contract, we first need to deploy it. This requires access to an Ethereum network and funds to pay for gas. For development and testing purposes, we used Ganache, a local Ethereum simulator that automatically provides ten test accounts, each preloaded with 100 ETH.

Ganache runs locally on port `7545`, as shown in Figure 1. This local server allows us to simulate blockchain transactions without using real cryptocurrency or connecting to a live network.

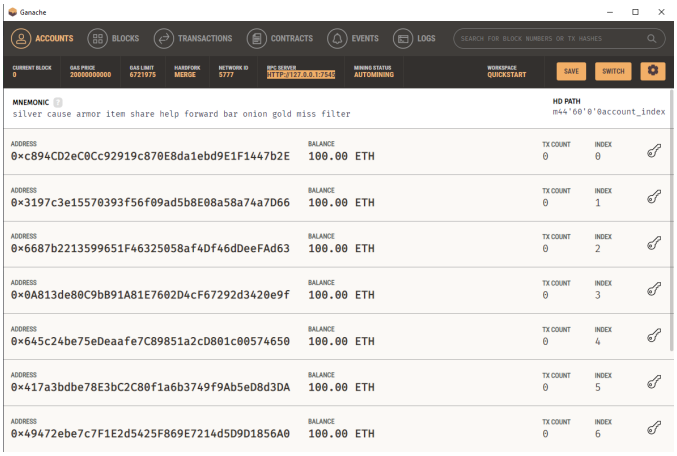

Fig. 1. Ganache user interface

This port is essential because, using the loopback address, we can connect Remix IDE to the local blockchain provided by Ganache.

For clarity during testing, we use only the first account (index 0) to deploy the contract. The remaining accounts are used exclusively to interact with the contract, such as invoking functions or modifying data.

IV. Deploying the Smart Contract

With our local server running and the smart contract ready, we proceed to deploy it. First, we compile the Solidity code, as shown in Figure 2.

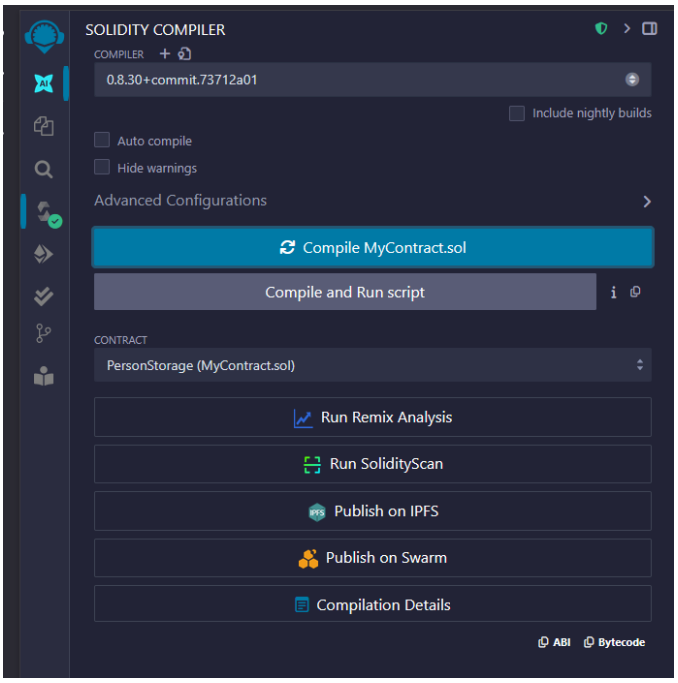

Fig. 2. Remix compiler with the smart contract loaded

After compilation, we go to the ”Deploy & Run Transactions” tab. By default, Remix is set to use a virtual machine, as seen in Figure 3. To connect to our local Ganache network, we select the `Custom - External Http Provider` option, shown in Figure 4.

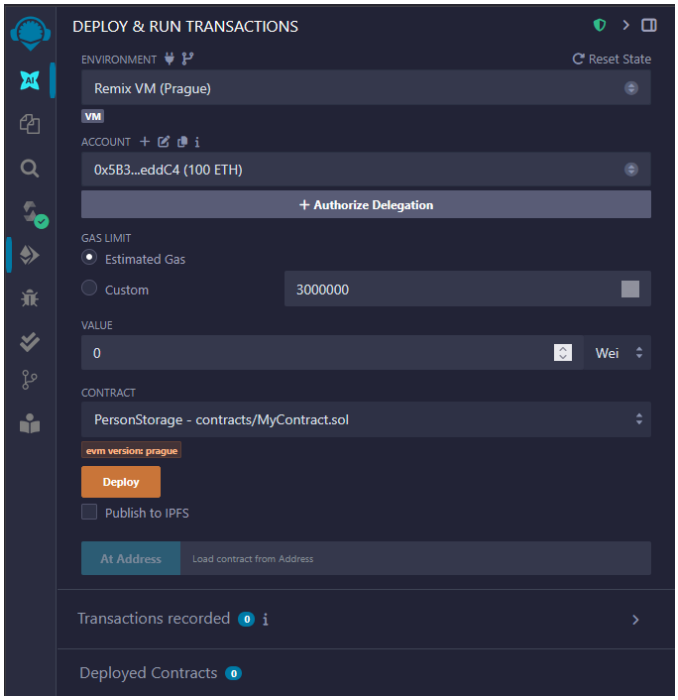

Fig. 3. Default Remix environment set to JavaScript VM

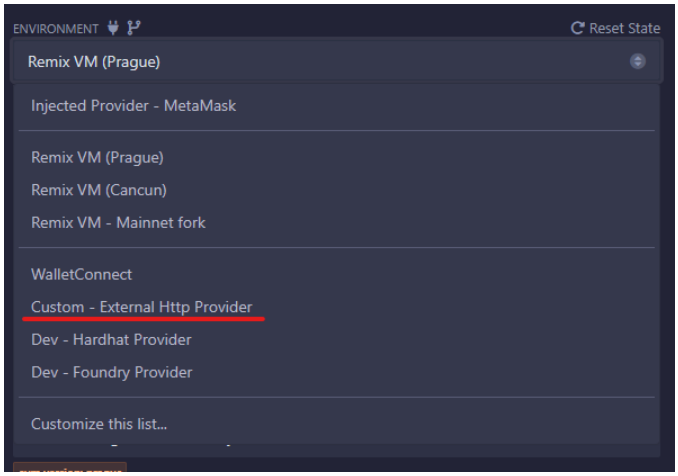

Fig. 4. Option to connect to a custom external HTTP provider

In the popup dialog, we enter the IP address and port of our local Ganache server, which is `127.0.0.1:7545`, as shown in Figure 5. This is the loopback address associated with our local test blockchain.

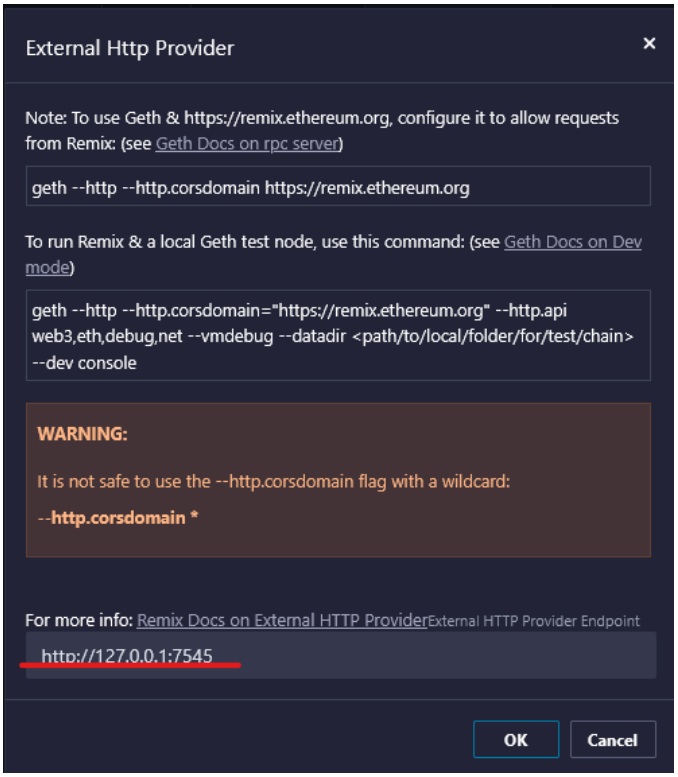

Fig. 5. Connecting Remix to Ganache via loopback address

We then ensure that the selected account is the first one (index 0) and proceed with the deployment, as shown in Figure 6.

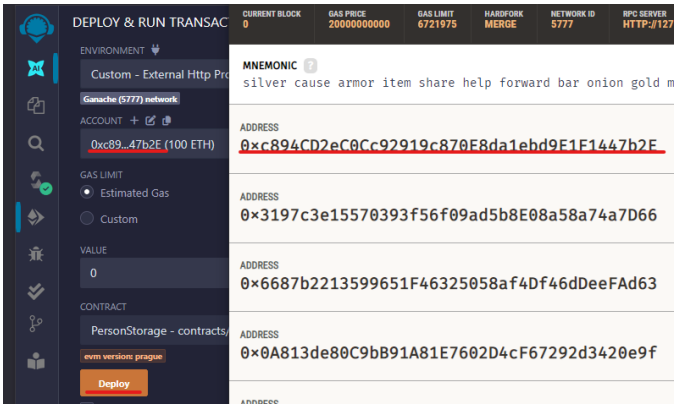

Fig. 6. Deployment of the smart contract using the first account

Once deployed, a new block is created in Ganache, and the address of the deployed contract can be found inside the transaction details, as shown in Figure 7[1].

[1]The sender address shown in the image may differ because Ganache was restarted. It still corresponds to account index 0 in the current session.

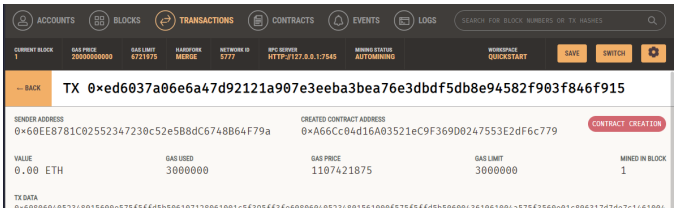

Fig. 7. Ganache transaction showing contract address

In this case, Remix did not automatically detect the deployed contract. However, by manually entering the contract address obtained from Ganache, we can confirm that the deployment was successful, as shown in Figure 8.

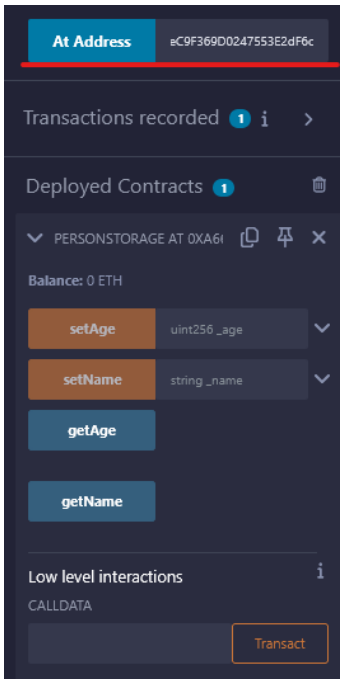

Fig. 8. Verifying the deployed contract manually via address

V. Connecting MetaMask with Ganache

To interact with our contract, we need to connect Ganache with MetaMask, which is a browser wallet that allows the use of custom networks. To do this, we add a custom network with the parameters shown in Figure 9.

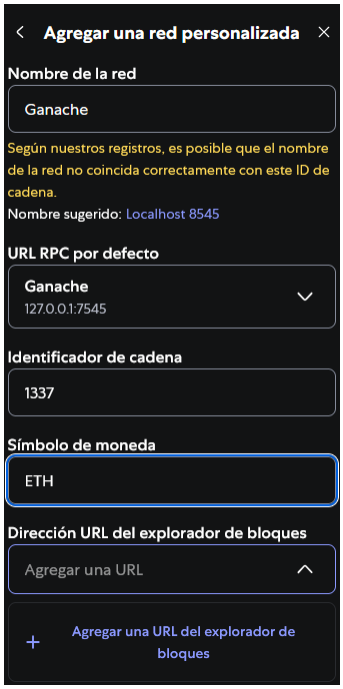

Fig. 9. Parameters for MetaMask network configuration

Next, we import the second account (index 1) into MetaMask using its private key, as shown in Figure 10. This key can be found in Ganache by clicking on the key icon.

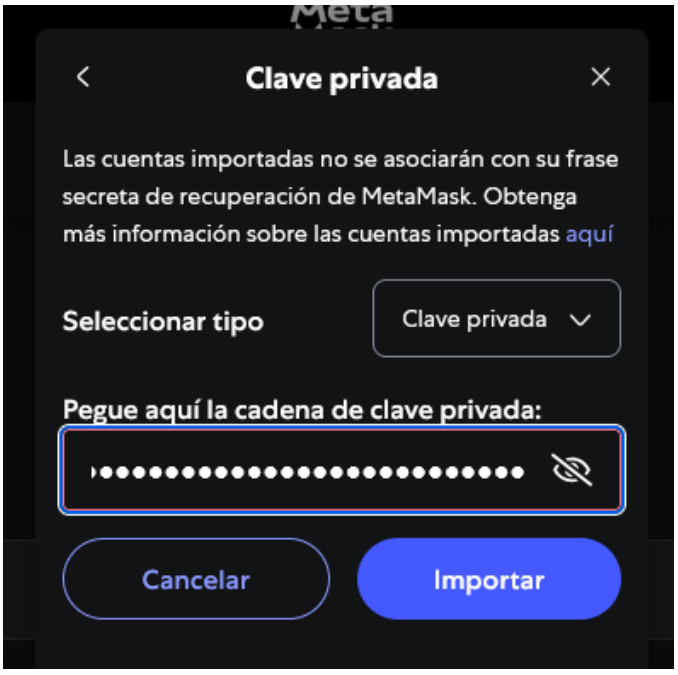

Fig. 10. Adding an account using a private key

Now we see our account having th 100ETH given by Ganache.

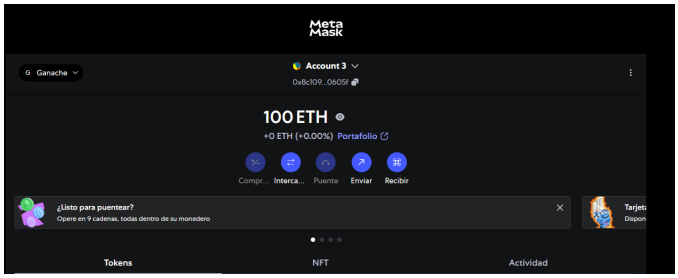

Fig. 11. Funds in account by connecting ganache

VI. Setting up the WebApp

For our web application, we use HTML, JavaScript, and some CSS for styling. The final interface is shown in Figure 12.

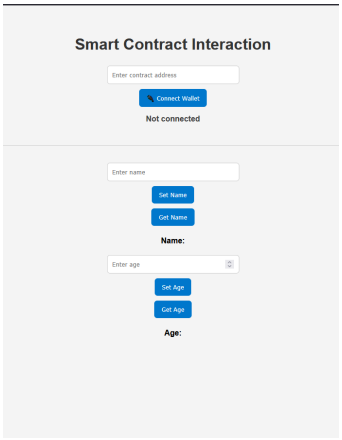

Fig. 12. Display of the WebApp

The HTML consists of a field to input the address of the smart contract and a button to connect to it. Then, there are two input groups, each with a button and a text box: one for setting the name and another for setting the age. Below these, there is a text display showing the stored information.

Since the HTML and CSS are relatively straightforward, we focus here on the JavaScript code, which contains the core functionality.

A. JavaScript Functionality

Our main library is `ethers.js`, a powerful tool for interacting with the Ethereum blockchain and its ecosystem. For the interaction to work, three key elements must be present:

- **Provider:** An abstraction to connect to the Ethereum network. It allows read-only access to the blockchain and its state.
- **Signer:** Represents an entity that can sign messages and transactions using a private key. It is required for performing operations that modify the blockchain.
- **Contract:** An abstraction that allows interaction with a deployed smart contract on the blockchain. It behaves like a normal JavaScript object. [1]

MetaMask provides both a **Provider** and a **Signer**—it connects to the Ethereum network and securely stores the user's private key.

Connecting to MetaMask with `ethers.js` is done as follows:

```
provider = new ethers.providers.Web3Provider(
    window.ethereum);
await provider.send("eth_requestAccounts", [])
    ;
signer = provider.getSigner();
```

Listing 2. Connecting to MetaMask using ethers.js

This snippet is executed (partially) when the user clicks the "Connect" button. The MetaMask popup appears as shown in Figure 13.

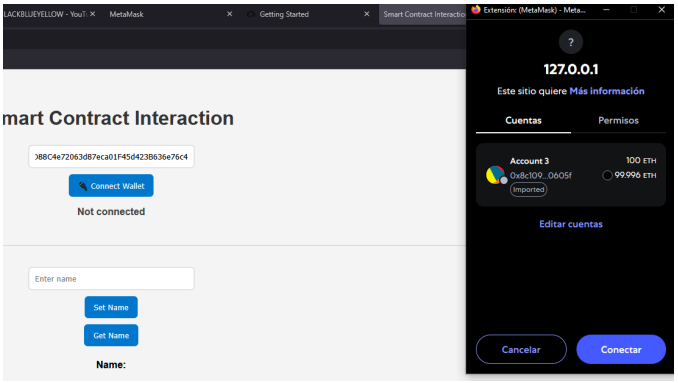

Fig. 13. MetaMask connection popup in the WebApp

Here, the user selects an account, which was previously imported using its private key.

Once connected, we create a contract instance using the contract address, its ABI, and the signer. The ABI is hardcoded in our JavaScript file and defines the structure of the smart contract, including variables and callable functions (as defined in Listing 1). This allows us to interact with the contract as if it were a JavaScript object.

```
contract = new ethers.Contract(inputAddress,
    abi, signer);
async function setName() {
  const name = document.getElementById("
      nameInput").value;
  try {
    const tx = await contract.setName(name);
    await tx.wait();
    alert("Name set successfully!");
  } catch (err) {
    console.error("Error setting name:", err);
    alert("Failed to set name");
  }
}
```

Listing 3. Creating and interacting with the contract

As shown, calling a smart contract function is as simple as invoking a method. This same pattern is used for setting the name and age values. When a "Set" button is pressed, MetaMask prompts the user to confirm the transaction, as seen in Figure 14.

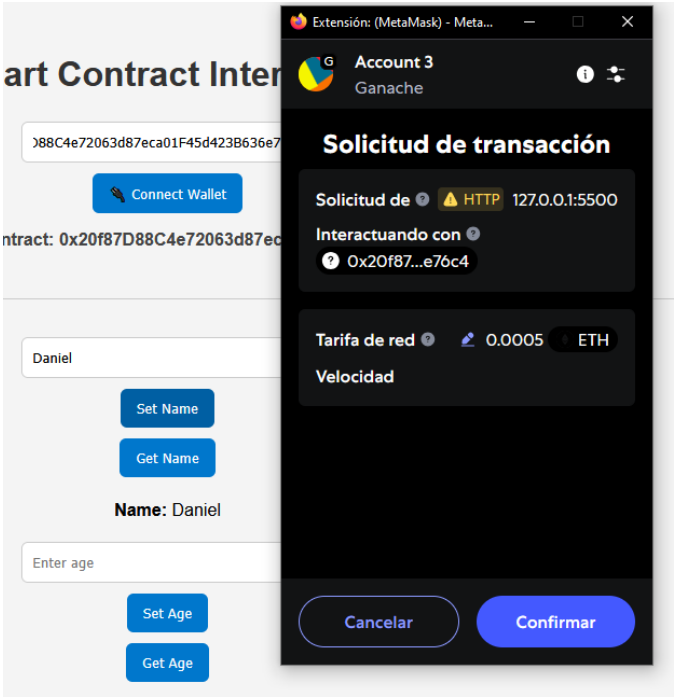

Fig. 14. MetaMask popup after pressing "Set Name"

After the user confirms, a new block is created and added to the blockchain. This can be verified in Ganache, as shown in Figure 15.

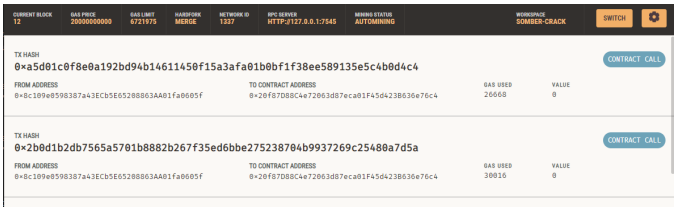

Fig. 15. Block associated with the transaction.

VII. Results

To verify that the web application interacts correctly with the deployed smart contract, we created a short demonstration video.

Video: Cybersec SmartContract Consumption Demo

The video illustrates the process of sending a transaction to the blockchain via the contract and confirms its inclusion in Ganache. It visually supports the results discussed in the previous section.

The source files are available in the Cybersec SmartContracts Project Repository.

VIII. Conclusions

Through the development of this project, we successfully demonstrated the deployment and consumption of a sim-

ple smart contract using a web interface connected to the Ethereum blockchain via MetaMask and `ethers.js`.

The integration between the smart contract and the front-end application was verified both visually and functionally, as shown in the demonstration video and confirmed via Ganache. The ability to set and retrieve personal data like name and age confirms the correct behavior of the contract and the interaction flow.

Furthermore, this project highlights the simplicity and power of modern web3 tools, enabling secure and user-friendly decentralized applications. The combination of local development tools like Remix and Ganache, with frontend libraries like `ethers.js`, offers a complete environment for testing and deploying Ethereum-based dApps.

Future improvements could include enhanced UI/UX, input validation, and deploying the smart contract to a testnet or mainnet for broader testing scenarios.

IX. Considerations

This project demonstrated that building a smart contract consumer interface can be relatively straightforward. However, several issues arose during the implementation that may affect others attempting to reproduce the results.

One issue was related to the REMIX IDE, as shown in Figure 8. The exact cause of the problem is unclear, but it also seemed to affect interactions with `Ganache`. Occasionally, the functions `setName` and `setAge` were not recognized by Ganache, which disrupted the testing process.

Another consideration involved network configuration. Although the Ganache UI displays the network ID as `5777`, MetaMask only allows connections to network ID `1337`. This mismatch initially resulted in empty account balances, preventing transactions from being sent. It remains uncertain whether changing the network ID in Ganache resolved the issue or if it was a coincidence, possibly related to connectivity between MetaMask and the blockchain.

References

[1] Ethers.js Contributors. (2024) Ethers.js v5 documentation. Accessed: Jul. 24, 2025. [Online]. Available: https://docs.ethers.org/v5/